VMM ships with some pretty useful built-in components and applications. VMM"s Atomic Generator is probably one of the most powerful ones, yet it's pretty basic. It can definitely help you generate a flow of random items but it was not intended for generation of sequences. A sequence (a.k.a scenario) is a set of items that have some sort of correlation between them. For example - a set of 10 transactions with incremental addresses, or a set of 3 packets where the first one is always short and the last one is always long.

VMM addresses the need for smart scenarios with the "VMM Scenario Generator" - a separate generator - but integrating this component will require some work to get it up and running smoothly with the rest of your environment. Man, if we could just use atomic generators to define and create scenario as well... Guess what? here's a quick way to do this in your test program, with no special preparations and no learning curve.

The idea here is to leverage the atomic generation architecture that we already have to drive our own sequences, and we do this by first shutting down the atomic generator (see here how to do this in a more stylish way). We then take an array of items that we prepared in advanced (i.e. the sequence) and inject these items one by one into the output channel of the atomic generator. As a matter of fact, atomic generators implicitly encourage this kind of hack by supplying the inject() function in their API. With this approach the entire environment stays intact, while we get what we want with no extra effort and no extra maintenance of unnecessary generators.

In this little example here we create a sequence of 10 transactions with incremental addresses. If you get the hang of it, you can easily create smarter sequences using the same technique.

```
{code}
program test;
...

// preparing the sequence
// 10 transactions, address incrementing by 1
trans seq1[];

task gen_seq1();
  seq1 = new[10];
```

```
  foreach (seq1[i]) begin
    seq1[i] = new();
    seq1[i].randomize with  {
        address == 1000 + i;
    };
  end
endtask

...

initial begin
  bit dropped;
  ...
  // need to stop automatic generation
  env.start();
  env.trans_gen.stop_xactor();
  ...
  // creating the sequence
  gen_seq1();

  // launching a non blocking thread that will
  // inject our sequence
  fork
    foreach(seq1[i]) begin
      env.trans_gen.inject(seq1[i]), dropped);
    end
  join_none
  ...

  env.run();

end

endprogram
{/code}
```

We'd love to hear your comments! email us at mail@thinkverification.com, or simply fill in your comment below.